

Parallel_Programming_HW4_Report

MapReduce

李浩榮 110062401

Introduction

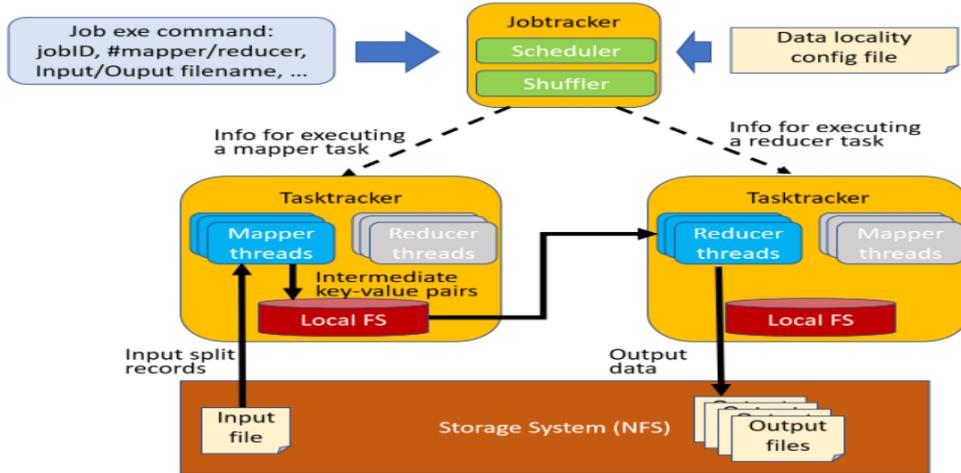
這次的作業主要是需要我們通過使用MapReduce的方式來完成Word-count的任務。讓我們更了解分散平行處理的重要性。目標是要找出給定的input file中的各個words的數量，並且正確輸出每組包含文字和數量的結果。

實作的流程主要是：

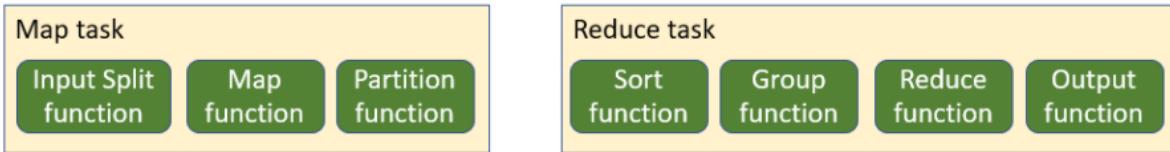
1. 讀取loc config file中標示的chunkID和nodeID為一個map的對應表。
2. 讀取input file中的所有行的文字，並且根據輸入的chunk_size參數去把文字行數做組合，組出有loc config file中 chunkID的個數。
3. 使用MPI根據給定的 node 參數，去開啟size個process。其中rank=0為jobtracker，其餘nodes皆為tasktracker。
4. tasktrackers nodes被建立後，主動通過 MPI_Send 向 jobtracker 發起 request (send的內容是自己的nodeID，以便jobtracker判斷)，並等待 jobtracker 判斷後發派下來的 chunk_data 才開始做字數計算等處理。
5. 同時 jobtracker 也會需要一直處於 while 迴圈中的 listening 狀態，根據 tasktracker 傳過來的自己的 nodeID，去比對 loc_file 中該 nodeID是否還有locality。
 - a. Have locality：如果有的話就指派對應的chunk data下去。
 - b. No locality：如果是loc_file上已經沒有自己的nodeID所對應的chunk需要被處理的話，那就可以去幫別人處理，但是該 tasktracker

必須先 sleep 個D秒(這裡會這麼做是為了模擬某些 data 是希望被通過同一個 nodeID 所處理, 理由是存取 local file system 比較快)

- c. No locality :如果是已經sleeped過了的 tasktracker, 接下來就可以直接去幫別人處理chunk data。
 - 6. 在 tasktracker 下會根據參數 CPUS 去開啟 CPUS 個 pthreads。每一個 threads 會根據該 tasktracker node 所被分發到的chunk data 平均地平行處理。
 - 7. 每個threads會讀取tasktracker下的local file system來檢查是否有出現過的文字以及數量, 一起做計算和更新。
 - 8. 同個 tasktracker 下所有 threads 處理的結果會先儲存在local file system。
 - 9. 所有threads處理完後便join起來, 關閉threads, 該tasktracker node回傳通知 jobtracker自己已經做好自己的部分了。並重新發起request去申請新的data 來處理。
 - 10. jobtracker會時時判斷是否所有的task都已經被領取完成且得到complete的回饋訊息, 若完成便直接跳出迴圈停止指派任務。
 - 11. tasktrackers們開始做reduce的動作, 合併相同的文字的數量, 並且回傳到唯一一個的reducer身上, 做sorting。
 - 12. 每個tasktracker的reducer都做好reduce後再使用MPI_Send回傳到 rank1 (其中一個)tasktracker上做最後的整合並且一起寫出到num_reducer個的.out 檔案中。
-
- map-reduce的架構



- map stage 和 reduce stage 所要做的任務:



- Data-locality 的概念圖

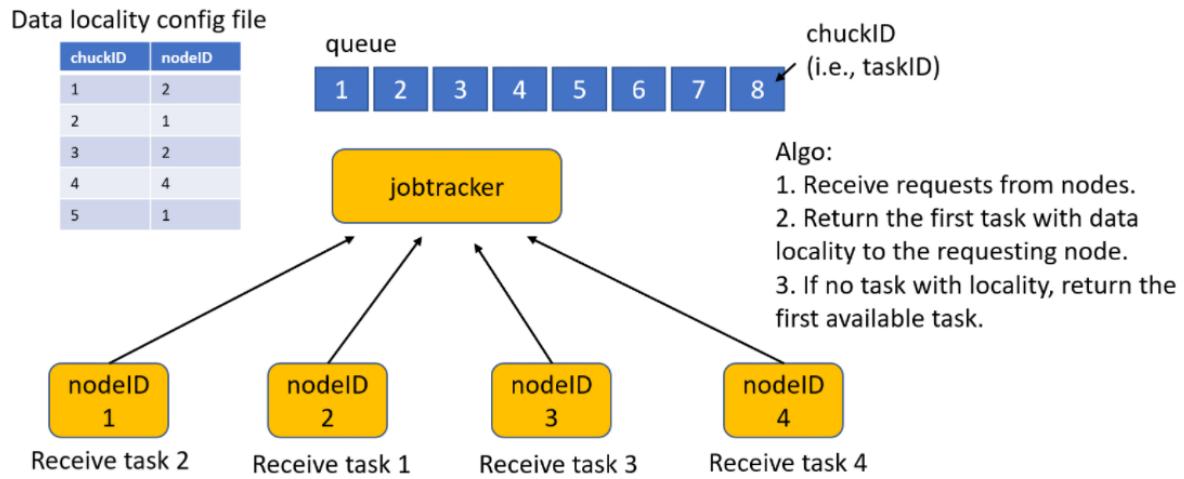


Fig. 3: Data locality-aware scheduling algorithm.

1. List the highlights of your implementation.

- 其實這次的作業技術來說不太難，真正的難點是要有很複雜的架構，例如 MPI 之間的溝通和 threads 之間的溝通。
- 比較有挑戰性的部分不外乎就是資料型態的轉換和 data 資料的運輸。因為 local threads 之間的溝通我是模擬實作了一塊 local file system 的空間來讓他們儲存結果。這很方便。
- 各個 tasktracker node 處理完的資料要 reduce 合併的部分都得用 MPI 傳輸，MPI 又不支援 string (我有試著找解法，但是好像要做很複雜的 casting，特別是我要傳輸的內容是 pair 的而且還是 string 和 int 不同型態的混合陣列 map)。礙於 MPI 的限制，我進行 iterate map 出來的結果轉化成 char* 去使用 MPI_Send, MPI_Recv 來實現進行不同 node 之間的溝通。
- 動態 (dynamic) 進行 jobtracker 和 request queue 以及 chunk 的分配。
- 和之前的作業 MPI+Pthread 不同的點在於，之前的只要計算完 mandebrot set 後便可以直接輸出圖片 array。但是這次的會有複數個不同 memory 地址的

local reduce的結果，不同process間 memory 也是invisible的，所以會出現不能直接global去讀其他process的variables或者計算結果之類的。甚至一些判斷的variables都需要通過MPI_Send, Recv 才能成功，相當繁瑣。

2. Detail your implementation of the whole MapReduce program?

- 主要的流程都在上述introduction的部分提到了。接下來會針對部分流程做詳細的條件解釋。
- 一開始的input_file中的文字和 loc_file 的 chunkID, nodeID 皆是以二維 vector (assign_chunk, loc_config) 儲存，主要是為了後續的access比對方便，才使用vector，速度會比較快。
- 每個chunkID會由一個陣列 task_available 來追蹤目前還可以被領取執行的 task chunk有哪一些。陣列會記錄該task chunkID 是否還沒被處理 (**READY_STATE**)，是已經被領取但是還沒完成(**TAKEN_STATE**)，已經被處理好(**FINISH_STATE**)。
- 另外也有一個陣列 tracker_doing_taskID來追蹤哪一些 task chunkID是被哪一個 tasktracker 拿去處理。
- MPI process node 中，我是以rank0當作 jobtracker 來做動作，rank1負責做最後整合所有 tasktrackers 的任務，自己本身也會處理chunk data，所以總共還是會有 size-1個tasktrackers。

```
if((myNodeID)%size-1 == tasktrackerID && task_available[myChunkID-1] == READY_STATE){
```

- 若有 tasktracker 發送 request 到 jobtracker 上，我會判斷tasktrackerID和 loc_config file中的取餘數後的nodeID是否一致，才進行分配。

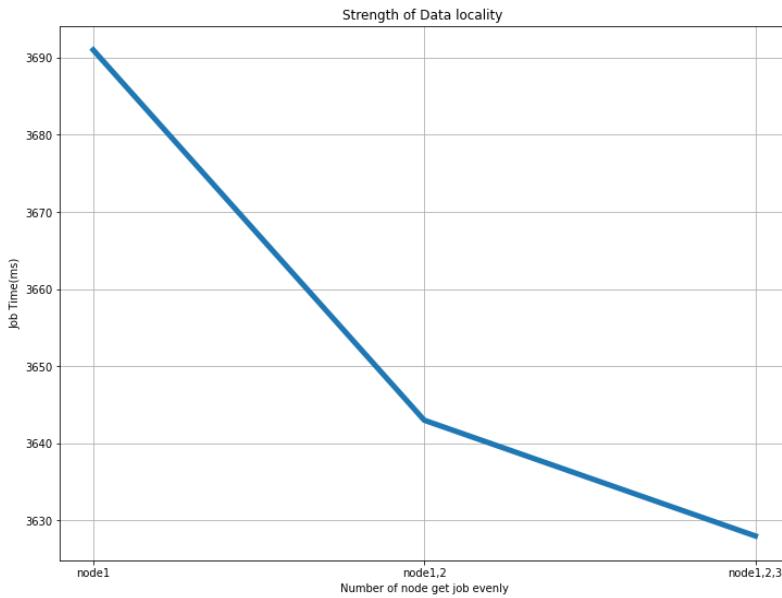
```
// no locality
// first time sleep
char sleeping_message[6] = "SLEEP";
// cout << tasktracker_sleeped[tasktrackerID-1] << endl;
if(tasktracker_sleeped[tasktrackerID-1] == 0){ //haven't sleep yet
    MPI_Send(sleeping_message, 6, MPI_CHAR, tasktrackerID, 0, MPI_COMM_WORLD);
    tasktracker_sleeped[tasktrackerID-1] = 1;
}
```

- sleep Delay的部分也是由 tasktracker_sleeped來追蹤哪一項tasktracker是已經被delay過的。

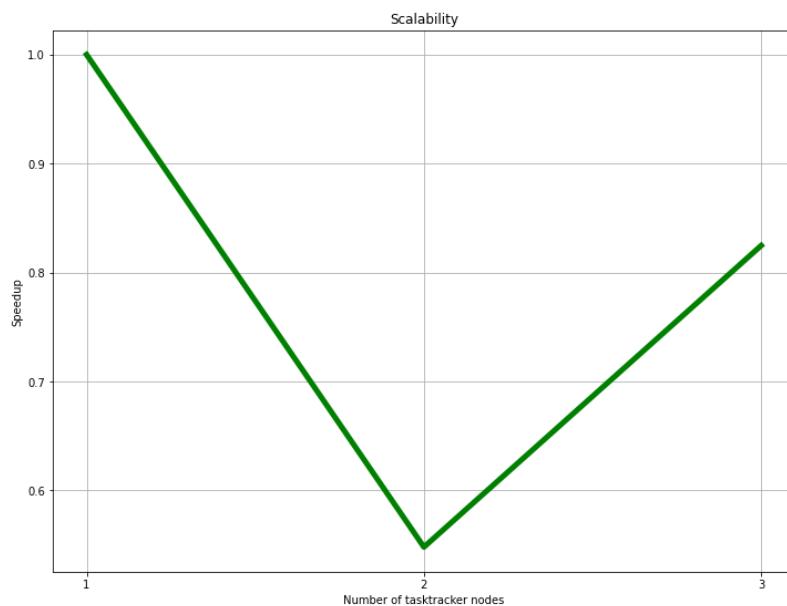
- 整個 rank=0 的區塊裡主要會有兩個while迴圈，第一個while迴圈會一直運轉，也就是出於listening狀態，負責Recv request from tasktracker，並且分配任務，隨時更新 task chunk 的狀態。每次loop都會檢查還有哪一些task還沒被領取，如果都被領取完了，那麼rank 0的 jobtracker就可以停止listening的指派動作。
- rank=0的另外一個while迴圈負責的是接收 tasktracker 完成被指派的任務後的通知訊息(使用不同的MPI tag做區隔)，如果有 chunkID數量的回報完成的話，就表示tasktracker們已經沒有東西可處理，這裡便會負責發送終止的訊息給tasktracker node們，離開他們rank!=0的區塊。
- 對於 rank !=0 的部分，都是tasktracker的工作內容流程。主要是Send request, recv data chunk，然後切割收到的recv_buf，接著打開 num_threads個線程，依序分配patch_data下去給threads們處理，結果存在 local_file_system。
- 當所有rank都完成上述的事情後，jobtracker處於待機狀態，rank1要負責 reduce回收其他rank combine reduce的結果。方法也是vector, map, array 的拆分或組合，使用MPI來溝通。
- 最後由rank1來sort 並且根據nnum_reducer輸出複數個.out檔案。

3. Conduct the experiments to show the performance impact of data locality, and the scalability.

- 為了方便比對，接下來的experiment都是基於testcase10的loc和word來做實驗，理由是testcase10的測資比較大，也比較容易看出時間上的差異。且 testcase10 用到了最大的四個node，和12個cpus，可以最大化各個設定之間的差異性。



- 首先對於 Strength of data locality的部分，我是改了 testcase10 的 loc 檔案的三個 tasktracker node 的 ID 分佈比例。node1 表示 nodeID 為 1 要負責約八成的 chunkID，其餘兩成分別給 node2, node3。（這裡是為了看出 locality，所以 node 的總數量不變，只是改變各自被分配的工作量）。
- node1,2 是指 node1 和 node2 個別拿了四成的工作量，剩下的給 node3。
- node1, 2, 3 是個別平均分配了各三成的工作量。
- 可以看到隨著每個 node 拿到的工作量平均後，交換不同 node 所帶來的 time cost 也明顯減少了。



- 這部分是讓testcase10只泡在複數個node下(亦即一個是jobtracker,其餘為tasktracker)的情況。
- 我這裡是為了觀測而把sleep的部分註解,因為一個node的話,它就不會進行sleep,隨著node的增加,每個sleep 3秒的話,speedup就沒有太大意義了。
- 這也可以觀察到使用兩個node(兩個tasktrackers)是比只有一個來的沒有效率的。可能是因為有不同node就會出現要sleep delay的情況,而且我們的這個word count計算速度很快,所以communication反而降低了整體時間。
- 但是三個nodes的speedup相對兩個nodes的則較有提升,我認為應該是MPI的communication cost被data locality所分擔掉所以才會開始出現上升趨勢,這也是這次作業的重點。

4. Commands

- `srun -N<NODES> -c<CPUS> ./mapreduce JOB_NAME NUM_REDUCER DELAY INPUT_FILENAME CHUNK_SIZE LOCALITY_CONFIG_FILENAME OUTPUT_DIR`

5. Experiments and conclusion

- **What have you learned from this homework?**
 - 從這次的作業我學會了遠距file system的分散平行處理的重點,也對於MPI+Pthread 的Hybird parallel programming有更進一步的認識。
 - C++ vector好用,速度快,但是泛用性有限,支援有限,形態轉變是bottleneck。
 - 資料有 locality的特性,搬對資料,且有事先標記,對於後續存取有很大的幫助。
 - 存手寫模擬建立整個不同device去讀storage system的資料來處理,並以 jobtracker 的 schedule 和 shuffler 來管控是有點小複雜,下次可以試著使用分散式的平台也許會更得心應手。
- **Feedback**
 - 這學期以來學到了很多平行程式的技術和概念。讓我對於演算法和程式執行時間有更深刻的概念。不得不說這堂課真的硬生生地提高了我寫作C和C++的程式能力,受益良多。
 - 老師的淳淳善誘的教導下,讓我們都可以非常仔細地學到很多平行的觀念,這是我非常推薦這堂課的原因。

- 最後也感謝助教這學期以來給與同學們的作業以及課堂疑問解答上的幫助，真的是非常辛苦了！謝謝！這是一堂很好的課。